CS 6410: Compilers

Fall 2019

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins. Today lecture notes are a modified version of his lecture notes.

Credits For Course Material

- Big thank you to UW CSE faculty member, Hal Perkins
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, …)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, …)

Administrivia

- My office hours:
  - On Mondays from 10:30-12pm in 401 Terry Ave, 142 classroom
  - By appointment
- Homework:
  - First homework – due on Saturday, October 5
  - Second homework – will be posted today
    - Due on Saturday, October 29
- Project setup:
  - First part – due on Saturday, October 12

Agenda

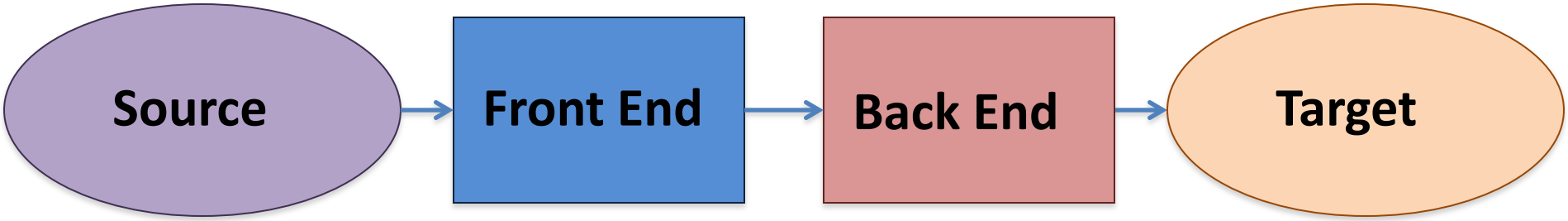
- Context free grammars
- Ambiguous grammars
- LR Parsing
- Table-driven Parsers

Reading: Cooper & Torczon 3.1-3.3

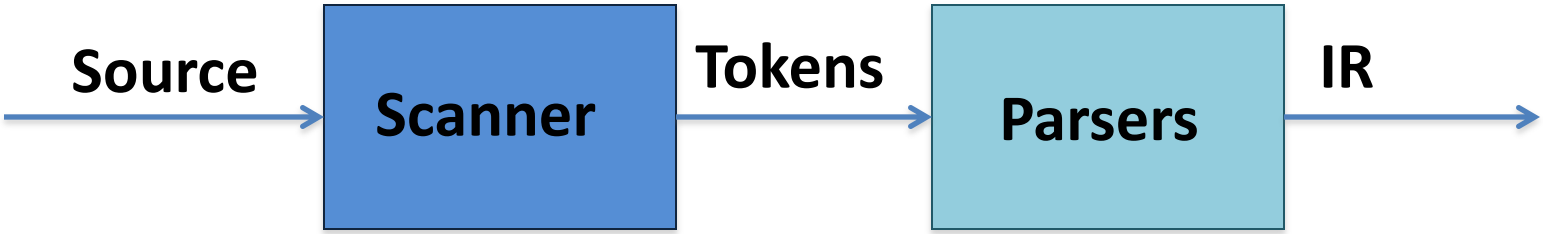
(The Dragon book is also particularly strong on grammars and languages)

Review: a Structure of a Compiler

- At a high level, a compiler has two pieces:
  - Front end – analysis
    - Read source program, and discover its structure and meaning
  - Back end – synthesis
    - Generate equivalent target language program



Review: Compiler - Front End



- Front end is usually split into two parts:
  1. Scanner – responsible for converting character stream to token stream: keywords, operators, variables, constants
     - Also: strips out white space, comments
  2. Parser - reads token stream; generates IR
     - Either here or shortly after, perform semantics analysis to check for things like type errors

Formal Languages & Automata Theory (One slide review)

- Alphabet: a finite set of symbols and characters
- String: a finite, possibly empty sequence of symbols from an alphabet
- Language: a set of strings (possibly empty or infinite)
- Finite specifications of (possibly infinite) languages
  - Grammar – a generator; a system for producing all strings in the language (and no other strings)
  - Automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings)

Backus-Naur Form (BNF)

- **Backus-Naur Form (BNF):** a syntax for describing language grammars in terms of transformation rules, of the form:

  <symbol> ::= <expression> | <expression> ... | <expression>

  - Terminal: a fundamental symbol of the language
  - Non-terminal: a high-level symbol describing language syntax, which can be transformed into other non-terminal or terminal symbol(s) based on the rules of the grammar

  - developed by two Turing-award-winning computer scientists in 1960 to describe their new ALGOL programming language

Finite State Automaton

- In other words, a finite state automaton is a formal mathematical object, defined as a five-tuple (S, Σ, δ, s_0, S_A):
  - S – the finite set of states of an automation (it may include an error state s_e)
  - Σ – the finite set of transition states (in the case of a scanner, the set of transition states is the alphabet)
  - $\delta(s, c)$ – the transition function that, for each state s, and each character (symbol) from the set $\{\Sigma \cup \varepsilon\}$ gives a set of new states
  - s_0 – the initial (start) state
  - S_A – the set of accepting (final) states

Finite State Automaton

- Based on their purpose, finite state automata (finite state machines) can be classified into three major groups:
  - Acceptors (recognizers) - automata that compute Boolean functions. They do so by either accepting or rejecting the inputs given to them
  - Classifiers – automata that has more than two final states and it gives a single output when it terminates
  - Transducers – automata that that produces outputs based on current input and/or previous state is called a transducer. Transducers can be of two types:
    - Mealy machine
    - Moore machine

DFA vs NFA

DFA	NFA
Deterministic –for every input symbol, a transition is from a current state to a particular next state	Nondeterministic – for at least one input symbol, a transition is from a current state to multiple possible next states
Empty ε transitions do not exist	Empty transitions are permitted
Backtracking possible	Backtracking is not possible
Typically, requires more space	Typically, requires less space
Some sequence of inputs is accepted, if the automaton transitions to a final state	Some sequence of inputs is accepted, if at least one of the possible transitions ends in a final state

Acceptability by DFA and NFA

- Some rules:
  - A string is accepted by a DFA/NFA if and only if the DFA/NFA starting at the initial state ends in an accepting state (any of the final states, for the NFA) after reading the whole string.
  - In other words, some string s ∈ S is accepted by a DFA/NFA (Q, Σ, δ, q0, F), if and only if $\delta^*(q0, S) \in F$

  - A language L accepted by DFA/NFA if $\{S | S \in \Sigma^* \text{ and } \delta^*(q0,S) \in F\}$

Removing Empty Transitions from Finite Automata

In there exists an empty transition between some nodes X and Y in a NFA, we can remove it as follows:

- Find all the outgoing arcs from Y.
- Copy all these arcs, starting from X, without changing the arcs' labels.
- If X is an initial state, make Y also an initial state.
- If Y is a final state, make X also a final state.

From NFA to DFA

- Subset construction algorithm:
  - Input: an NFA
  - Output: an equivalent DFA

1. Create state table from the given NFA
2. Create a blank state table under possible input alphabets for the equivalent DFA.
3. Mark the start state of the DFA the same as NFA, q_0
4. For every possible input, find the combination of states that can be reached from the current state. Those states form a new DFA state, {Q0, Q1,... , Qn}
5. Every time a new DFA state is generated under the input alphabet columns, repeat step 4 again, otherwise go to step 6
6. The states which contain any of the final states of the NFA are the final states of the equivalent DFA.

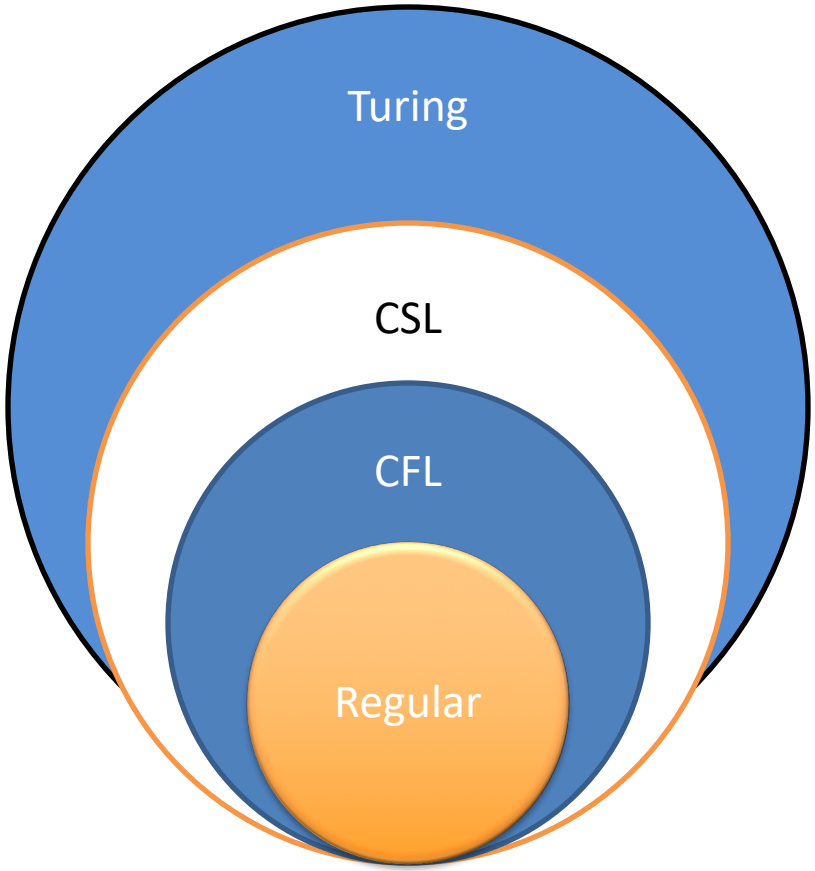
Syntactic Analysis

Syntactic Analysis / Parsing

- Goal: Convert token stream to an abstract syntax tree
- Abstract syntax tree (AST):
  - Captures the structural features of the program
  - Primary data structure for next phases of compilation
- Plan
  - Study how context-free grammars specify syntax
  - Study algorithms for parsing and building ASTs

Language (Chomsky) hierarchy: quick reminder

- Regular (Type-3) languages are specified by regular expressions/grammars and finite automata (FSAs)
  - Specs and implementation of scanners
- Context-free (Type-2) languages are specified by context-free grammars and pushdown automata (PDAs)
  - Specs and implementation of parsers
- Context-sensitive (Type-1) languages … aren't too important (at least for us)
- Recursively-enumerable (Type-0) languages are specified by general grammars and Turing machines



Context-free Grammars

- The syntax of most programming languages can be specified by a context-free grammar (CGF)
- Compromise between
  - REs: can't nest or specify recursive structure
  - General grammars: too powerful, undecidable
- Context-free grammars are a sweet spot
  - Powerful enough to describe nesting, recursion
  - Easy to parse; restrictions on general CFGs improve speed
- Not perfect
  - Cannot capture semantics, like "must declare every variable" or "must be int" – requires later semantic pass
  - Can be ambiguous (something we'll deal with)

Derivations and Parse Trees

- Derivation: a sequence of expansion steps, beginning with a start symbol and leading to a sequence of terminals
- Parsing: inverse of derivation
  - Given a sequence of terminals (aka tokens) recover (discover) the non-terminals and structure, i.e., the parse tree (concrete syntax)

Example

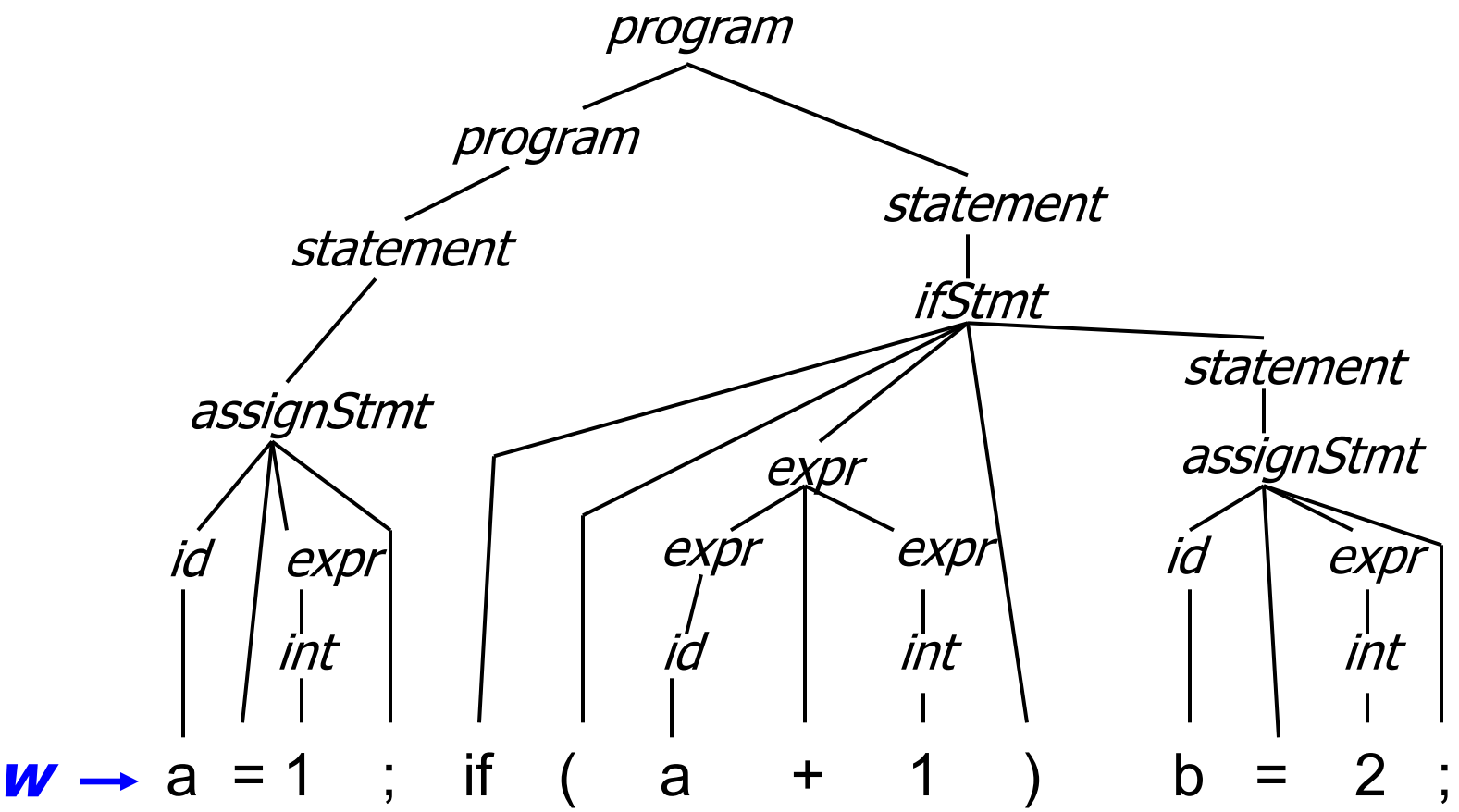
G

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Example

G

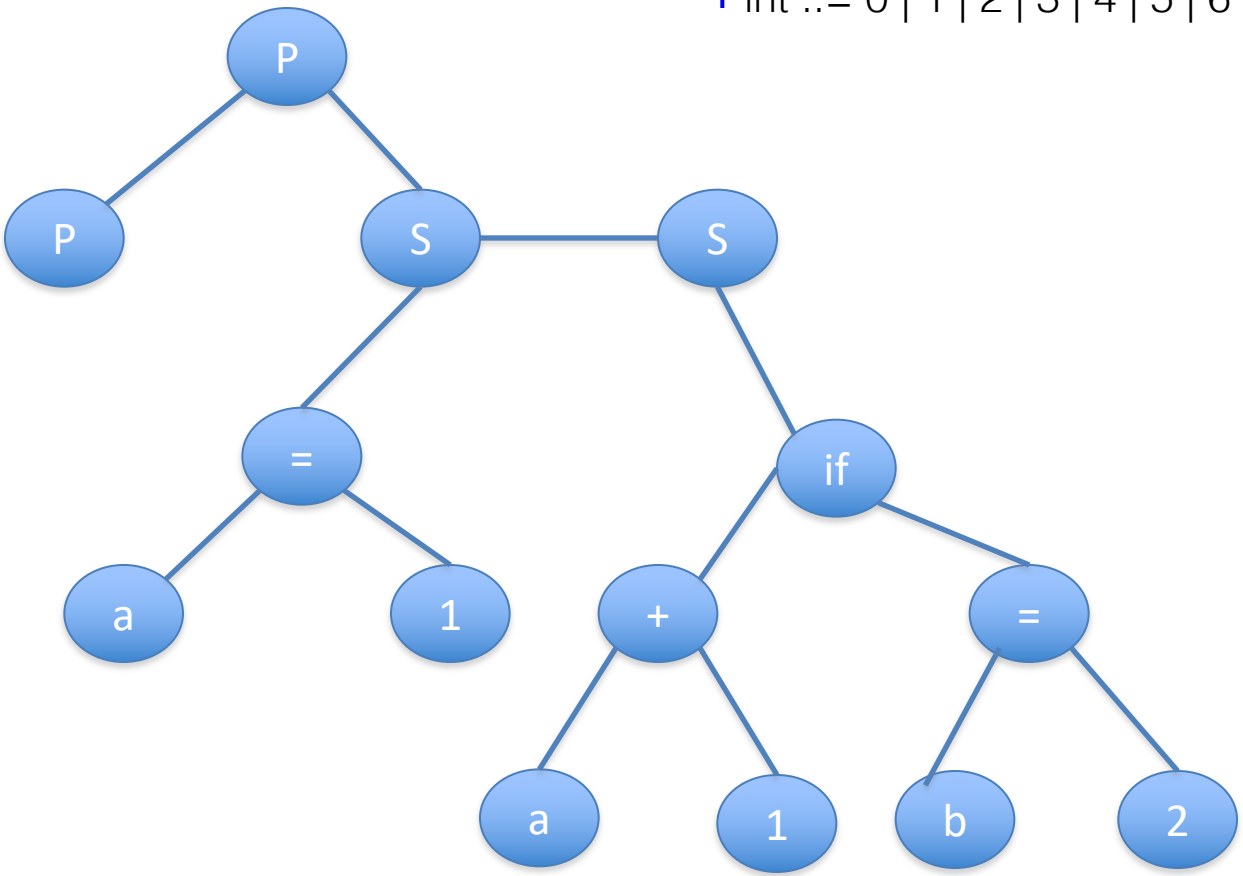
program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Example

G

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



a = 1 ; if (a + 1) b = 2 ;

Parsing

- Parsing: Given a grammar *G* and a sentence *w* in *L(G)*, traverse the derivation (parse tree) for *w* in some *standard order* and do *something useful* at each node
  - The tree might not be produced explicitly, but the control flow of the parser will correspond to a traversal

"Standard Order"

- For practical reasons we want the parser to be *deterministic* (no backtracking), and we want to examine the source program from *left to right*.
  - (i.e., parse the program in linear time in the order it appears in the source file)

Common Orderings

- Top-down
  - Start with the root
  - Traverse the parse tree depth-first, left-to-right (leftmost derivation)
  - LL(k), recursive-descent
- Bottom-up
  - Start at leaves and build up to the root
    - Effectively a rightmost derivation in reverse(!)
  - LR(k) and subsets (LALR(k), SLR(k), etc.)

"Something Useful"

- At each point (node) in the traversal, perform some semantic action
  - Construct nodes of full parse tree (rare)
  - Construct abstract syntax tree (AST) (common)
  - Construct linear, lower-level representation (often produced in later phases of production compilers by traversing initial AST)
  - Generate target code on the fly (done in 1-pass compilers; not common in production compilers)
    - Can't generate great code in one pass, – but useful if you need a quick 'n dirty working compiler

Context-Free Grammars

- Formally, a *grammar G* is a tuple $<N, \Sigma, P, S>$ where
  - N is a finite set of *non-terminal* symbols
  - Σ is a finite set of *terminal* symbols (alphabet)
  - P is a finite set of *productions*
    - A subset of $N \times (N \cup \Sigma)^*$
  - S is the *start symbol*, a distinguished element of N
    - If not specified otherwise, this is usually assumed to be the non-terminal on the left of the first production

Standard Notations

a, b, c elements of Σ (terminals)

w, x, y, z elements of Σ^* (zero or more terminals)

A, B, C elements of N (non-terminals)

X, Y, Z elements of $N \cup \Sigma$ (terminal or non-terminal)

α, β, γ elements of $(N \cup \Sigma)^*$ (zero or more terminals and non-terminals)

$A \rightarrow \alpha$ or $A ::= \alpha$ if $<A, \alpha> \in P$

Derivation Relations (1)

- $\alpha\ A\ \gamma \Rightarrow \alpha\ \beta\ \gamma$ iff $A ::= \beta$ in *P*
  - derives
- $A \Rightarrow^* \alpha$ if there is a chain of productions starting with A that generates α
  - transitive closure

Derivation Relations (2)

- $w\ A\ \gamma \Rightarrow_{lm} w\ \beta\ \gamma$ iff $A ::= \beta$ in *P*
  - derives leftmost
- $\alpha\ A\ w \Rightarrow_{rm} \alpha\ \beta\ w$ iff $A ::= \beta$ in *P*
  - derives rightmost
- We will only be interested in leftmost and rightmost derivations – not random orderings

Languages

- For A in *N*, *L*(A) = { w | A =>* w }
- If *S* is the start symbol of grammar *G*, define
  L(*G*) = *L*(*S*)
  - Nonterminal on left of first rule is taken to be the start symbol if one is not specified explicitly

Reduced Grammars

- Grammar *G* is *reduced* if and only if (iff) for every production A ::= α in *G* there is a derivation

  S =>* x A z => x α z =>* xyz

  – i.e., no production is useless
- Convention: we will use only reduced grammars
  – There are algorithms for pruning useless productions from grammars – see a formal language or compiler book for details

Ambiguity

- Grammar *G* is *unambiguous* iff every *w* in *L*(*G*) has a unique leftmost (or rightmost) derivation
  - Fact: unique leftmost or unique rightmost implies the other
- A grammar without this property is *ambiguous*
  - Note that other grammars that generate the same language may be unambiguous, i.e., ambiguity is a property of grammars, not languages
- We need unambiguous grammars for parsing

Example: Ambiguous Grammar for Arithmetic Expressions

expr ::= *expr* + *expr* | *expr* - *expr*
| *expr* * *expr* | *expr* / *expr* | *int*

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

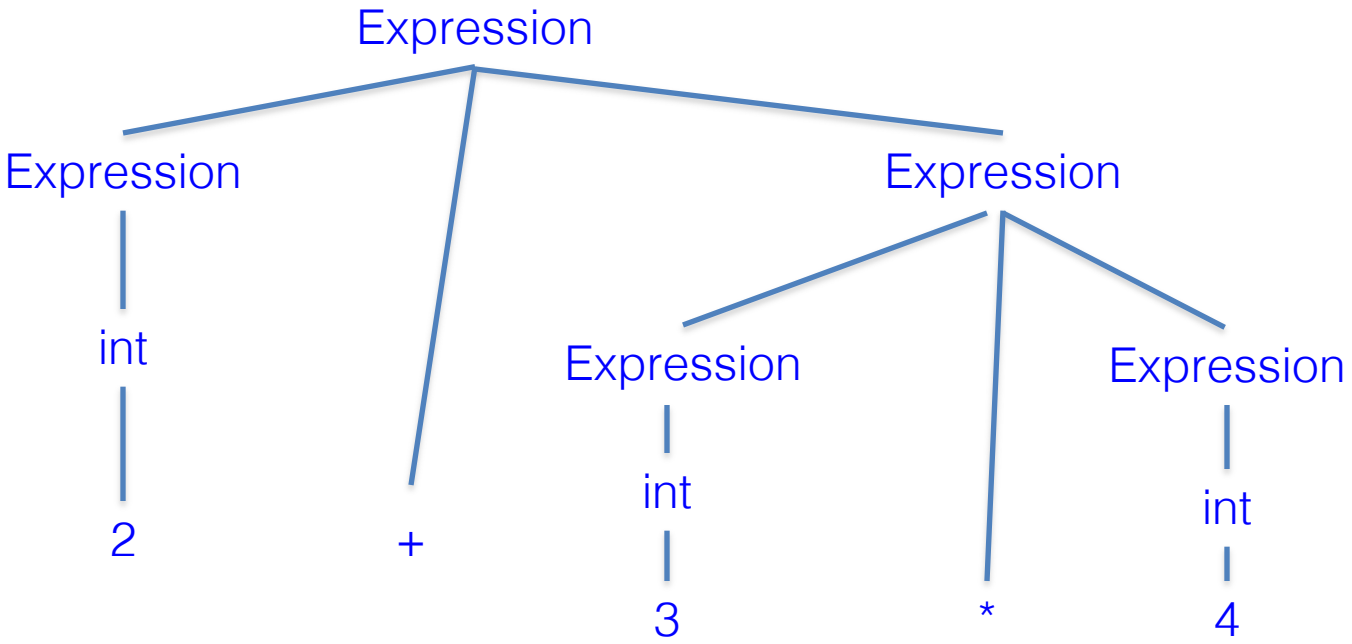
- Exercise: show that this is ambiguous
  - How? Show two different leftmost or rightmost derivations for the same string
  - Equivalently: show two different parse trees for the same string

expr ::= *expr* + *expr* | *expr* - *expr*
| *expr* * *expr* | *expr* / *expr* | *int*

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Example (cont)

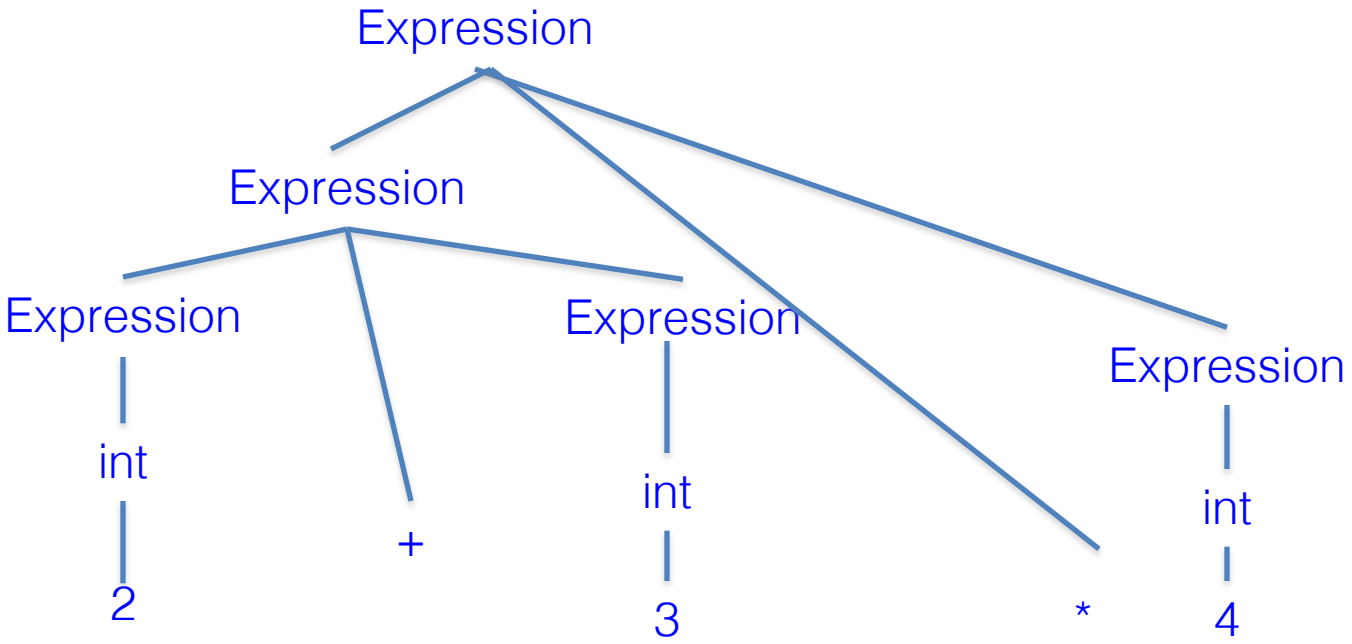
- Give a leftmost derivation of 2+3*4 and show the parse tree



expr ::= *expr* + *expr* | *expr* - *expr*
| *expr* * *expr* | *expr* / *expr* | *int*
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Example (cont)

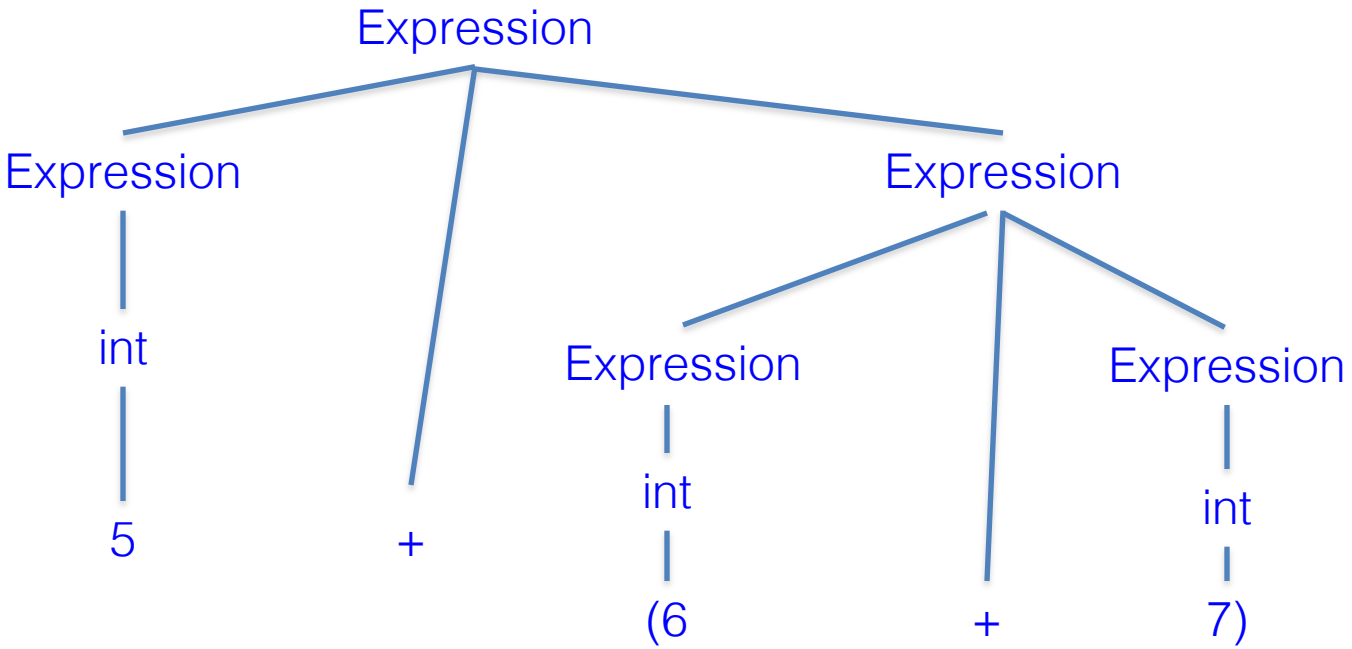
- Give a different leftmost derivation of 2+3*4 and show the parse tree



expr ::= *expr* + *expr* | *expr* - *expr*
| *expr* * *expr* | *expr* / *expr* | *int*
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Another example

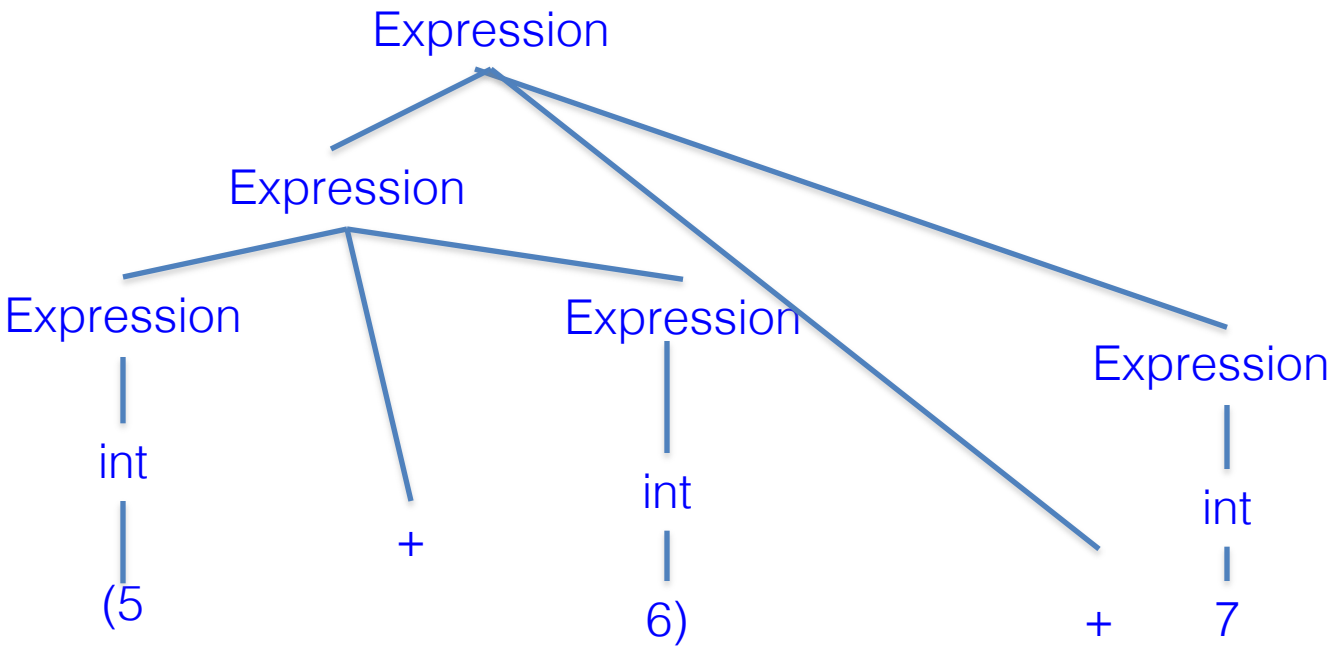
- Give two different derivations of 5+6+7



expr ::= *expr* + *expr* | *expr* - *expr*
| *expr* * *expr* | *expr* / *expr* | *int*
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Another example

- Give two different derivations of 5+6+7



What's going on here?

- The grammar has no notion of:
  - Precedence
  - Associatively
- Traditional solution
  - Create a non-terminal for each level of precedence
  - Isolate the corresponding part of the grammar
  - Force the parser to recognize higher precedence sub-expressions first
  - Use left- or right-recursion for left- or right-associative operators (non-associative operators are not recursive)

Classic Expression Grammar

(first used in ALGOL 60)

expr ::= *expr* + *term* | *expr* – *term* | *term*

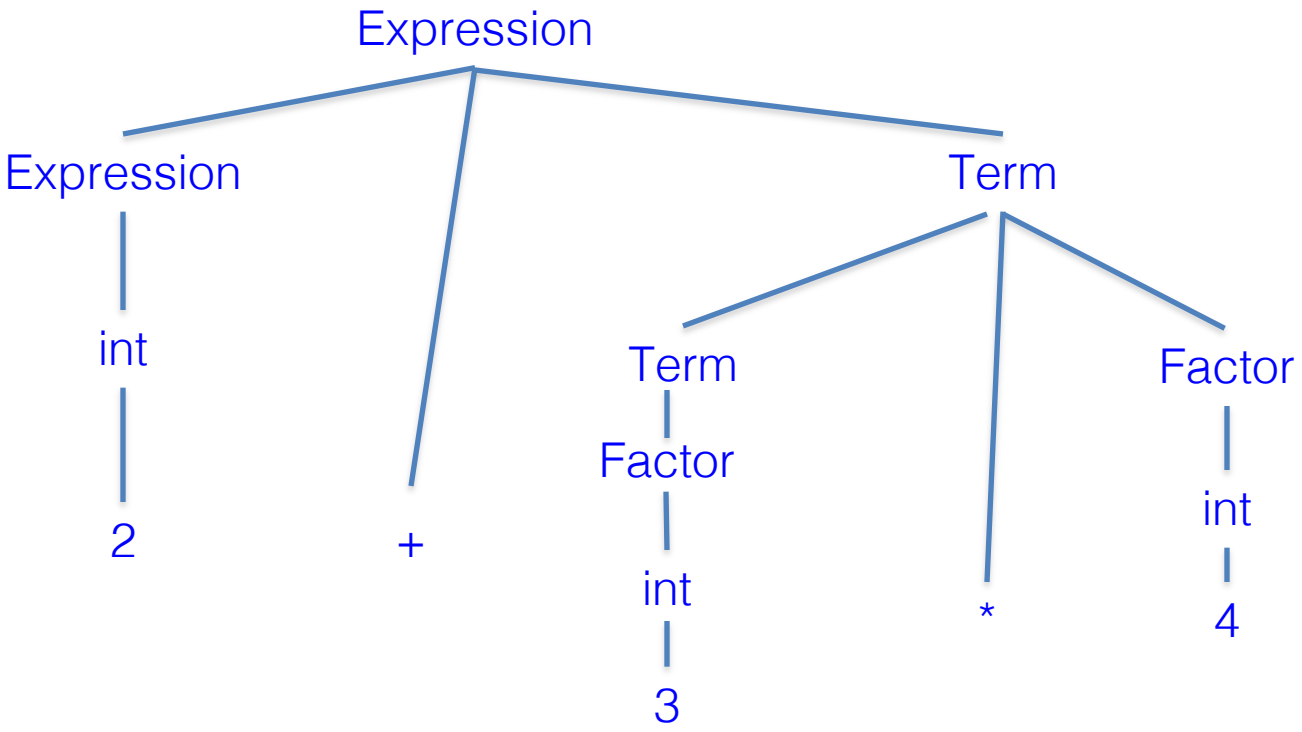
term ::= *term* * *factor* | *term* / *factor* | *factor*

factor ::= *int* | (*expr*)

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

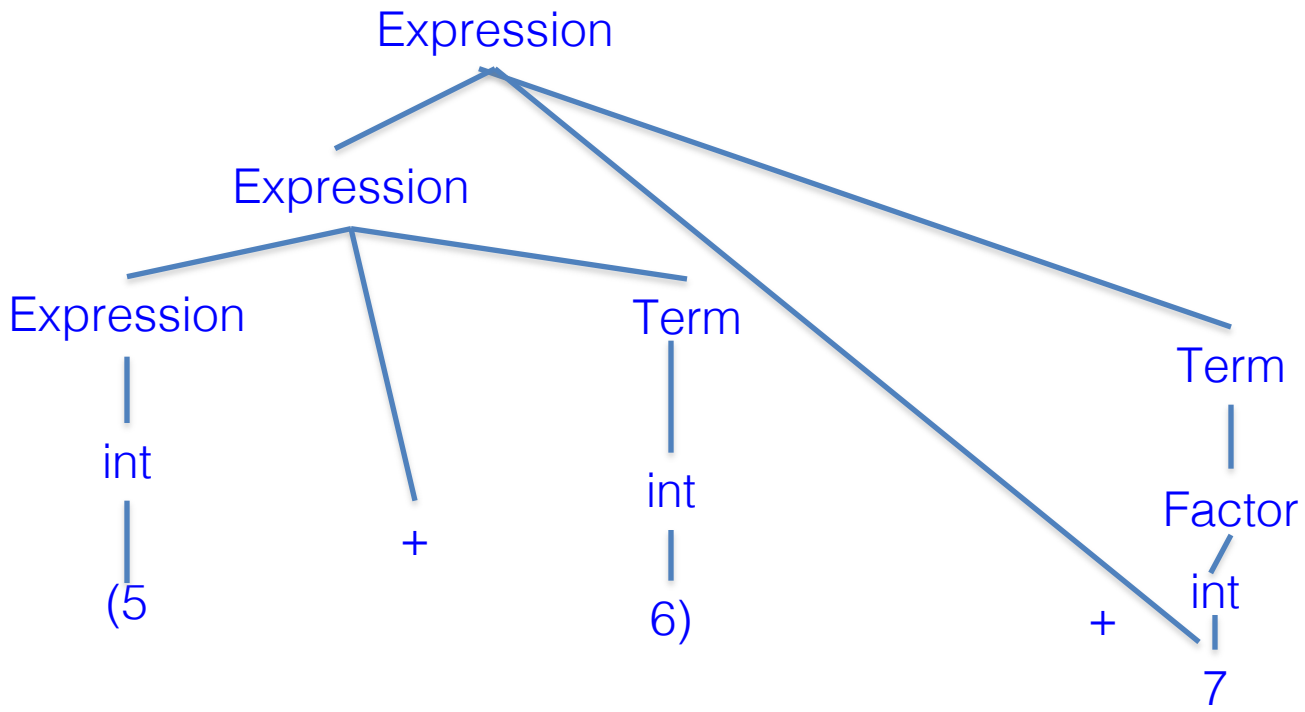
expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | (*expr*)
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Check: Derive 2 + 3 * 4



expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | (*expr*)
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

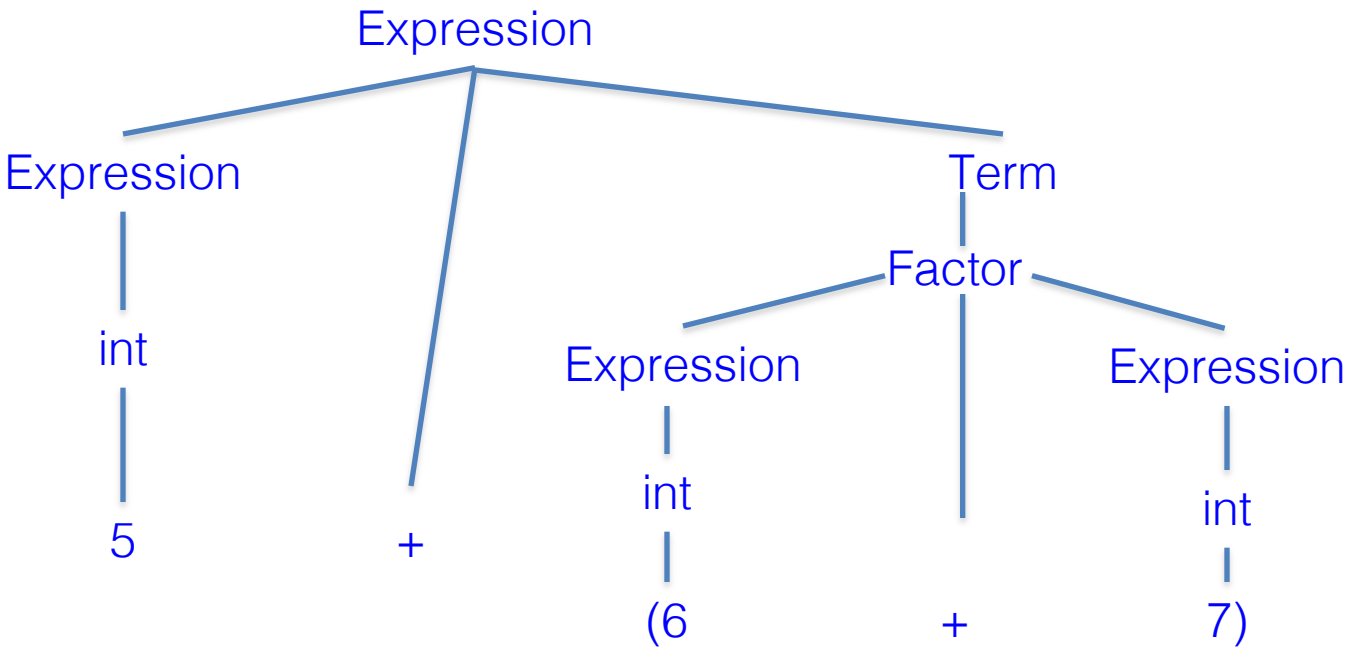
Check: Derive 5 + 6 + 7



- Note interaction between left- vs right-recursive rules and resulting associativity

expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | (*expr*)
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Check: Derive 5 + (6 + 7)



Another Classic Example

- Grammar for conditional statements

  stmt ::= if (*expr*) *stmt*

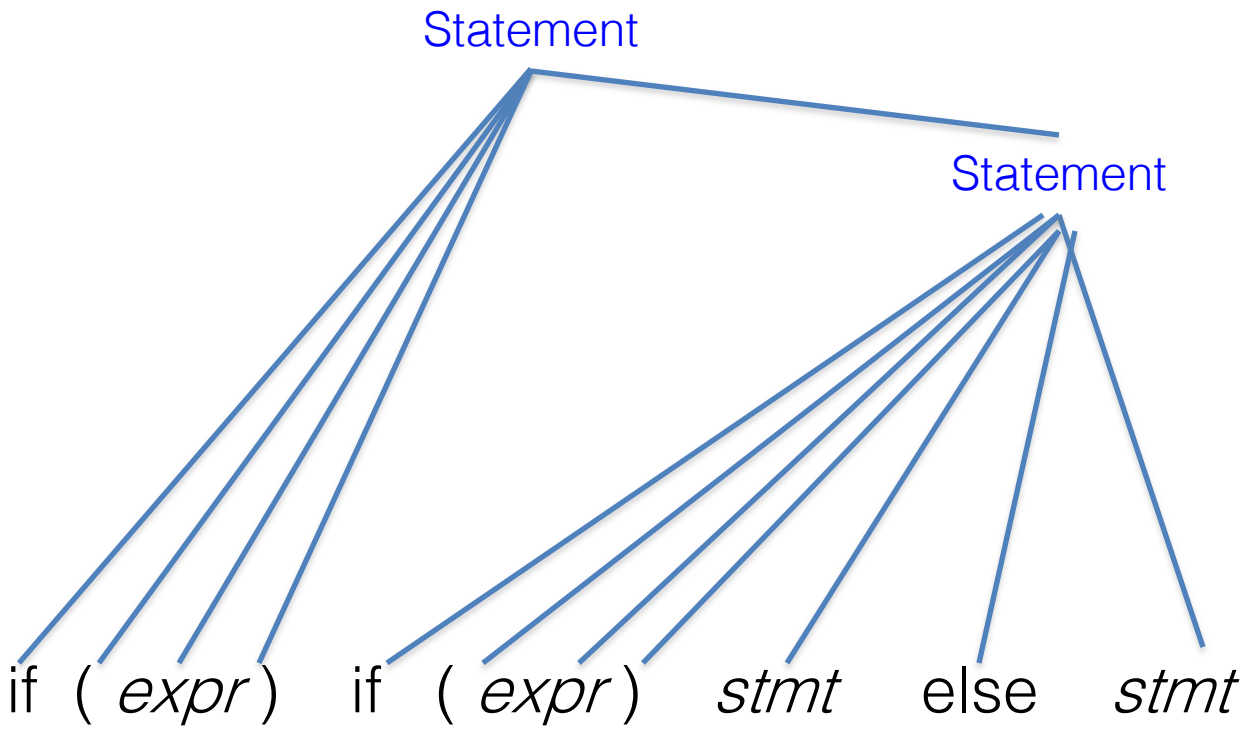
  | if (*expr*) *stmt* else *stmt*

  (This is the "dangling else" problem found in many, many grammars for languages beginning with Algol 60)

  – Exercise: show that this is ambiguous
    - How?

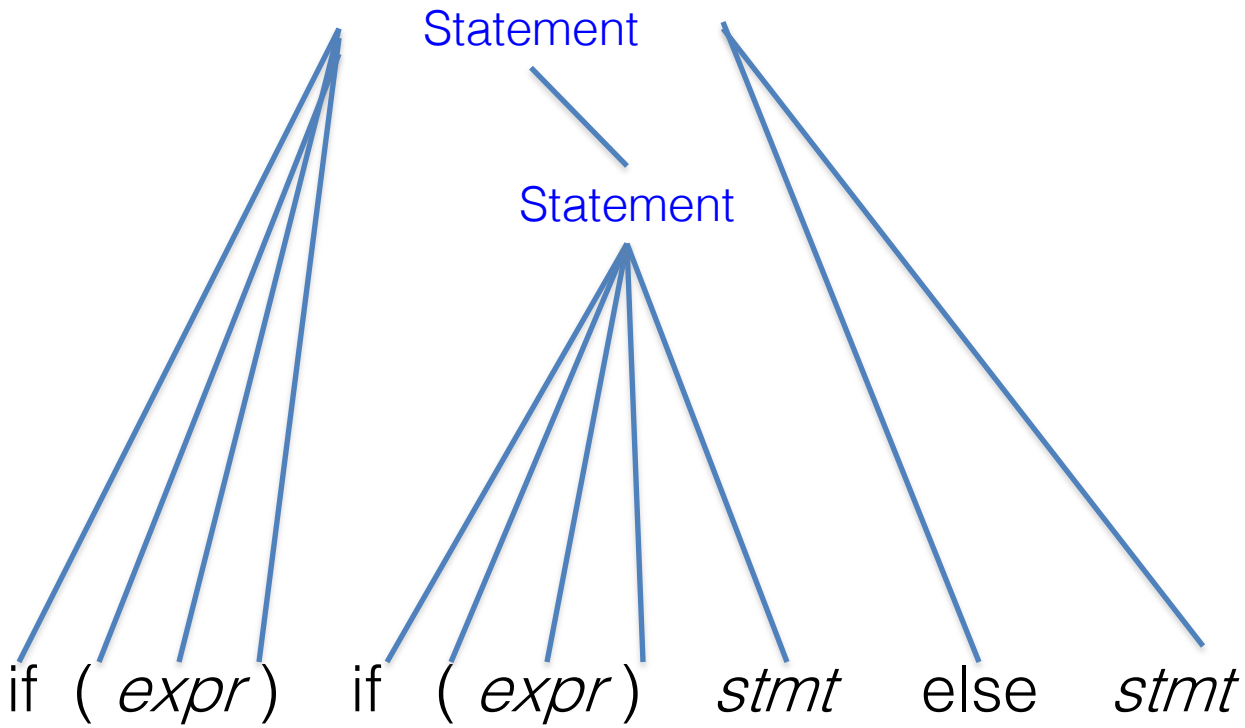
stmt ::= if (*expr*) *stmt*
| if (*expr*) *stmt* else *stmt*

One Derivation



stmt ::= if (*expr*) *stmt*
| if (*expr*) *stmt* else *stmt*

Another Derivation



Solving "if" Ambiguity

- Fix the grammar to separate if statements with else clause, and if statements with no else
  - Done in Java reference grammar
  - Adds lots of non-terminals
- Change the language
  - But it'd better be ok to do this – you need to "own" the language or get permission from owner
- Use some ad-hoc rule in the parser
  - "else matches closest unpaired if"

Resolving Ambiguity with Grammar (1)

Stmt ::= MatchedStmt | UnmatchedStmt

MatchedStmt ::= ... |

if (Expr) MatchedStmt else MatchedStmt

UnmatchedStmt ::= … |

if (Expr) Stmt |

if (Expr) MatchedStmt else UnmatchedStmt

– formal, no additional rules beyond syntax
– can be more obscure than original grammar

Resolving Ambiguity with Grammar (2)

- If you can (re-)design the language, just avoid the problem entirely

```
Stmt ::= ... |
    if Expr then Stmt end |
    if Expr then Stmt else Stmt end
```

  - formal, clear, elegant
  - allows sequence of Stmts in then and else branches, no { , } needed
  - extra end required for every if
    (But maybe this is a good idea anyway?)

Parser Tools and Operators

- Most parser tools can cope with ambiguous grammars
  - Makes life simpler if used with discipline
- Usually can specify precedence & associativity
  - Allows simpler, ambiguous grammar with fewer nonterminals as basis for parser – let the tool handle the details (but only when it makes sense)
    - (i.e., expr ::= expr+expr | expr*expr | … with assoc. & precedence declarations can be the best solution)

Parser Tools and Ambiguous Grammars

- Possible rules for resolving other problems:
  - Earlier productions in the grammar preferred to later ones (some danger here if grammar changes)
  - Longest match used if there is a choice (good solution for dangling if)
- Parser tools normally allow for this
  - But be sure that what the tool does is really what you want
    - And that it's part of the tool spec, so that v2 won't do something different (that you *don't* want!)

Parsers

Bottom-Up Parsing

- Idea: Read the input left to right
- Whenever we've matched the right hand side of a production, reduce it to the appropriate non-terminal and add that non-terminal to the parse tree
- The upper edge of this partial parse tree is known as the *frontier*

Example

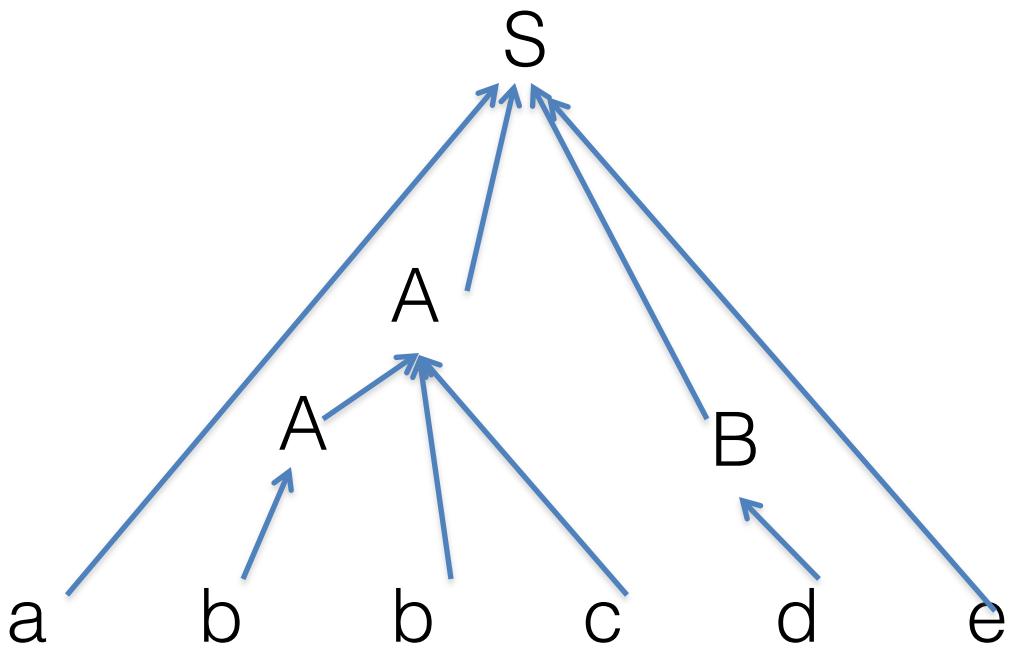
- Grammar

S ::= aAB e

A ::= Abc | b

B ::= d

- Bottom-up Parse



LR(1) Parsing

- We'll look at LR(1) parsers
  - Left to right scan, Rightmost derivation, 1 symbol lookahead
  - Almost all practical programming languages have a LR(1) grammar
  - LALR(1), SLR(1), etc. – subsets of LR(1)
    - LALR(1) can parse most real languages, tables are more compact, and is used by YACC/Bison/CUP/etc.

LR Parsing in Greek

- The bottom-up parser reconstructs a reverse rightmost derivation
- Given the rightmost derivation

  $S => \beta_1 => \beta_2 => \ldots => \beta_{n-2} => \beta_{n-1} => \beta_n = w$

  the parser will first discover $\beta_{n-1} => \beta_n$, then $\beta_{n-2} => \beta_{n-1}$, etc.
- Parsing terminates when
  - β_1 reduced to S (start symbol, success), or
  - No match can be found (syntax error)

How Do We Parse with This?

- Key: given what we've already seen and the next input symbol (the lookahead), decide what to do.
- Choices:
  - Perform a reduction
  - Look ahead further
- Can reduce $A \Rightarrow \beta$ if both of these hold:
  - $A \Rightarrow \beta$ is a valid production, *and*
  - $A \Rightarrow \beta$ is a step in *this* rightmost derivation
- This is known as a *shift-reduce parser*

Sentential Forms

- **Definition 2 – Sentential form:**

  If $S =>^* \alpha$, the string α is called a *sentential form* of the grammar

- In the derivation

  $S => \beta_1 => \beta_2 => \ldots => \beta_{n-2} => \beta_{n-1} => \beta_n = w$

  each of the β_i are sentential forms
- A sentential form in a rightmost derivation is called a right-sentential form (similarly for leftmost and left-sentential)

Handles

- Informally: **a handle -** a production whose right hand side matches a substring of the tree frontier *that is part of the rightmost derivation of the current input string*

  (i.e., the "correct" production)
  - Even if $A ::= \beta$ is a production, it is a handle only if β matches the frontier at a point where $A ::= \beta$ was used in *this specific* derivation
  - β may appear in many other places in the frontier without designating a handle
- Bottom-up parsing is all about finding handles

Handle Example

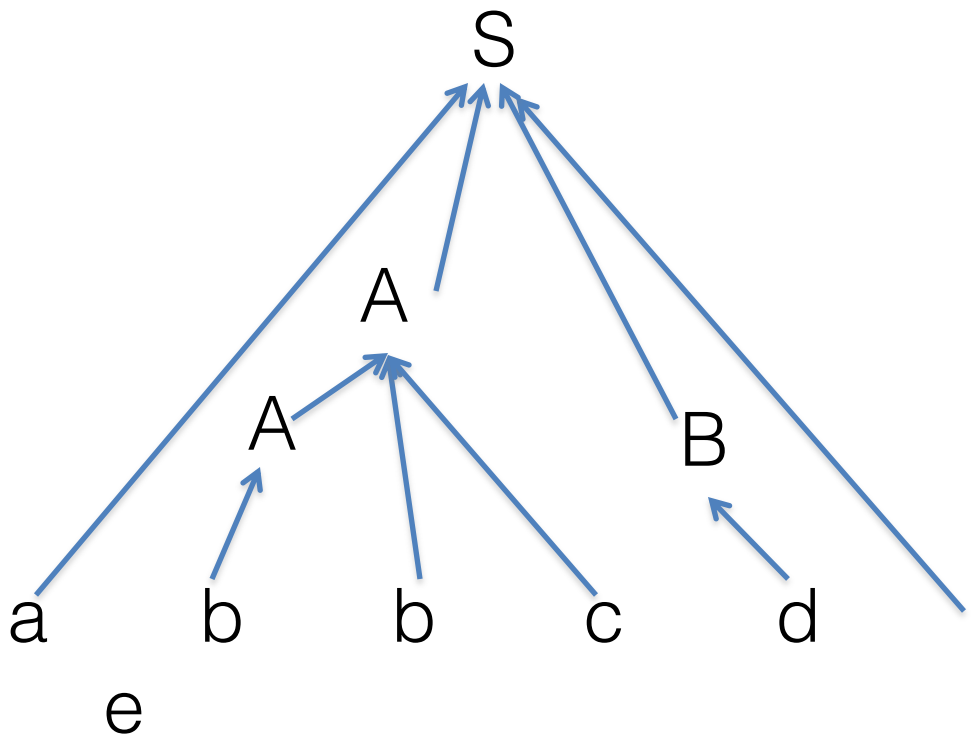
- Grammar

S ::= aAB e

A ::= Abc | b

B ::= d

- Bottom-up Parse



Handle Examples

- In the derivation

  S => a*AB*e => a*A*de => a*A*bcde => abbcde

  – abbcde is a right sentential form whose handle is *A*::=b at position 2

  – a*A*bcde is a right sentential form whose handle is *A*::=*A*bc at position 4

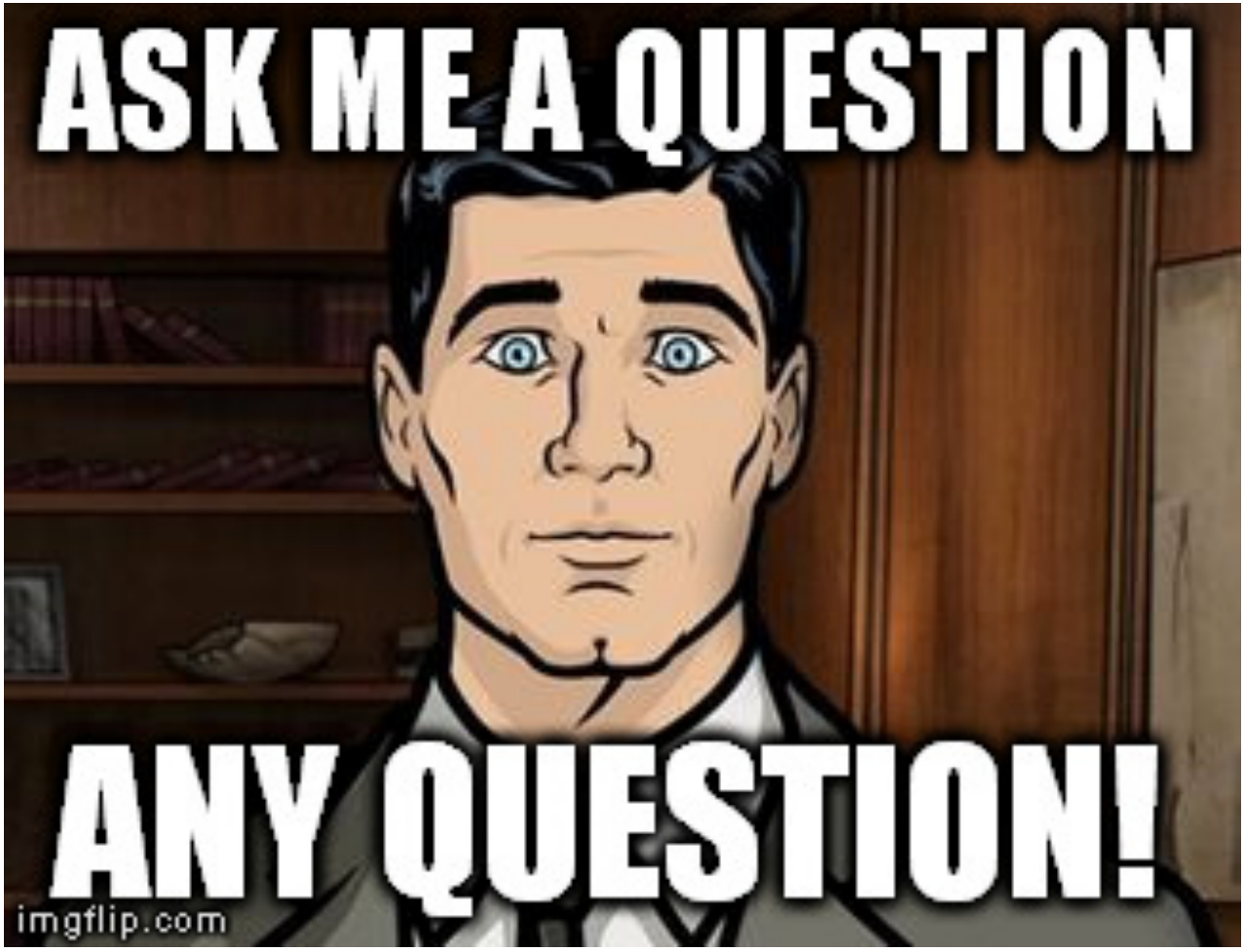
    - Note: some books take the left end of the match as the position

Handles – The Dragon Book Definition

- Definition 3 – Handle:
- Formally: a *handle* of a right-sentential form γ is a production $A ::= \beta$ and a position in γ where β may be replaced by A to produce the previous right-sentential form in the rightmost derivation of γ

Coming Attractions

- Constructing LR tables
  - We'll present a simple version (SLR(0)) in lecture, then talk about adding lookahead and then a little bit about how this relates to LALR(1) used in most parser generators



[Meme credit: imgflip.com]